アルゴリズム 62－1
（1998．5．20）

単一始点最短路問題 (SSSP) 線形時間アルゴリズムの実際的評価

東京大学大学院理学系研究科情報科学専攻
浅野泰仁, 今井浩

平成10 年 4 月 23 日

Abstract

単一始点最短路問題 (SSSP) を解くためのアルゴリズムとしては、Dijkstra のアルゴリズムが有名である。過去、Dijkstra のアルゴリズムを高速化する研究が多く行われてきたが、ソート問題に相当するボトルネックのため、線形時間を達成することはできなかった

1997 年、M . Thorup が整数枝重み無向グラフでの SSSP を線形時間で解くアルゴリズムを発表した。しかしこのアルゴリズムで使用されている複雑なデータ構造のいくつかは理論通りには実装できない。

本研究では、Thorup のアルゴリズムを現在の計算機上で実装するための変更を提案した上で、実際に Thorup のアルゴリズムの実装をおこなった。さらに、既存のアルゴリズムとの比較実験および各部分の実行時間計測をおこなった。

Practical evaluation of a linear-time algorithm for the single source shortest path problem (SSSP)

ASANO Yasuhito, IMAI Hiroshi
Department of Information Science, University of Tokyo

Abstract

SSSP is one of the well known classic problems in graph theory and Dijkstra's algorithm for SSSP is also quite popular. Several improvements of Dijkstra's algorithm have been studied, however, they could not accomplish a linear-time owing to its sorting bottleneck.

In 1997, M.Thorup proposed a linear-time algorithm for the SSSP on undirected and integer edge weight graph.

However, we can not implement this algorithm naively on computers today since the data structures used in the algorithm need a word of huge length.

We propose modifications to implement Thorup's algorithm and implement this algorithm. Moreover, compare execution times of the implementation and famous algorithms.

1 イントロダクション

単一始点最短路問題 (SSSP) はグラフ上のある 1 点 (始点) から、他のすべての点への最短距離を求める問題である。SSSP はグラフ理論の古典的問題のひとつであるが、多くの探索問題がその基礎として SSSP を含んでいるので、大規模な探索問題の基礎として SSSP の高速化は重要である。

SSSP を解くためのアルゴリズムとしては、Dijkstra のアルゴリズムが有名である。過去、ヒープ、プライオリティ・キューなどを用いて Dijkstra のアルゴリズムを高速化する研究が多く行われてきたが、Dijkstra

のアルゴリズムは本質的にソート問題に相当するボトルネックを含んでいるので、線形時間を達成することはできなかった (ここで「線形時間」とは計算機のワード長 b を含まない。すなわち、ラディックスソートなどは使用できない)。

1997年、M . Thorup によって整数枝重み無向グラフでの SSSP を $O(m)$ で解くアルゴリズムが発表された ([Tho97]) 。このアルゴリズムは、Dijkstra のボトルネックを回避するための新しい概念を導入し、MST の線形時間アルゴリズム ([FW94]) や過去に研究されてきたデータ構造などを利用して線形時間を達成している。しかしこのアルゴリズムで使用されている複雑なデータ構造のいくつかは理論通りには実装できない。

本研究では、Thorup のアルゴリズムを現在の計算機上で実装するための変更を提案した上で、実際に Thorup のアルゴリズムの実装をおこなった。さらに、既存のアルゴリズムとの比較実験および各部分の実行時間計測をおこない、Thorup のアルゴリズムの現在の実用性を評価した。結論としては、Thorup のアルゴリズムの実装は現在の計算機上ではフィボナッチヒープを用いた Dijkstra のアルゴリズムの実装より低速だったが、実行時間の大部分を MST の線形時間アルゴリズムが占めていること、使用しているデータ構造の再利用可能性などの有益な事実も得られた。

2 Thorup の線形時間アルゴリズム

Thorup のアルゴリズムを説明するために、その基本的な概念と、使用されているデータ構造の役割を述べる。その後、Thorup のアルゴリズムの概略を説明する。なお、仮定として、グラフ G は連結無向グラフであり、その枝の長さは 1 ワードに収まる正の整数である。従ってワード長を b とすると、枝長 $\ell(e) < 2^b$ となる。また、RAM を使用しているので、$\log n \leq b$ が前提である。また、$D(v)$ とは Dijkstra のアルゴリズムでも用いられる、$D(v) \geq d(v)$ ($d(v)$ は始点 s から点 v への実際の最短距離) を満たす、super distance と呼ばれるものである。

2.1 基本的な概念

Thorup は、コンポーネントとそのハイアラーキーという概念を導入した。グラフ G のコンポーネント $[v]_i$ とは、G の枝のうち長さが 2^i 未満のものだけを残したグラフの連結成分のうち、点 v を含むものである (なお、G が無向グラフであることに注意)。ハイアラーキーのための親子関係は、$[v]_i$ が $[w]_{i-1}$ の親 $\Leftrightarrow w \in [v]_i$ となる。なお、ハイアラーキーのルートは $[s]_b$ で、レベル 0 のコンポーネント $[v]_0$ は $\{v\}$ となる。このハイアラーキー上で、ルートから始めて全てのコンポーネントを visit することで、SSSP を解く。

コンポーネントを導入する目的は、Dijkstra のアルゴリズムのボトルネックの回避である。すなわち、$D(v) = d(v)$ のためのより柔軟な条件を得ることである。そのために、Thorup は、以下の定義を利用した。

$[v]_i^-$: $[v]_i \setminus S$.

min-child : $[v]_i$ が $[v]_{i+1}$ の *min-child* $\Leftrightarrow \min D([v]_i^-) \downarrow i = \min D([v]_{i+1}^-) \downarrow i$.

minimal : $[v]_i$ が *minimal* $\Leftrightarrow i \leq j \leq b-1$ について, $[v]_j$ が $[v]_{j+1}$ の $min-child$.

この min-child の定義からわかるように、ハイアラーキーのレベル i では、D の右 i ビットの誤差 (2^i 未満) を切り捨てて考えるというのが基本的な概念である。これは、もし $[v]_i$ と $[w]_i$ が異なるコンポーネントであるならば、定義より、$[v]_i$ と $[w]_i$ の距離は 2^i 以上だからである。

Thorup は、minimal なコンポーネントを visit していけば良いことを証明した。この条件は、上の定義からわかるように、D の右 i ビットの誤差を切り捨てているので、厳密に D が最小のものでなくても成り立つ。したがって、Dijkstra のアルゴリズム (厳密に D が最小のものを $D(v) = d(v)$ の条件とする) より、柔軟な条件と言える。

データ構造を用いた具体的なアルゴリズムは、次の項で述べる。

2.2 データ構造

2.2.1 コンポーネントツリー $\mathcal{T}$

コンポーネントツリー $\mathcal{T}$ は、コンポーネントハイアラーキーから不要な (同一の) コンポーネントを除去したものである。従ってハイアラーキーと同じように使うことができ、しかもそのサイズ (コンポーネント総数) は高々$2n$ となる。この $\mathcal{T}$ を線形時間 ($O(m)$) で構築する際に、MST の (RAM 使用) 線形時間アルゴリズム ([FW94]) を使って構築した MST を使う必要がある。なお、MST の構築が終われば、$\mathcal{T}$ の構築は、$O(n)$ でできる。

2.2.2 バケット

バケットは、前述の min-child を $\min D$ を比較せずに調べるために必要なデータ構造である。レベル 0 以外のコンポーネントは、それぞれ $\min D\downarrow i-1$ の取りうる値のインデックスを持つバケットを持っている。この $\min D\downarrow i-1$ の取りうる値の範囲が、バケットのサイズになる。これは、そのコンポーネントの diameter から用意に計算することができる。証明は省くが、Thorup は、MST から構築されたコンポーネントを用いることによって、バケットの総数は高々$8n$ になることを示した。なお、おのおののコンポーネントのバケットのサイズは、$\mathcal{T}$ の構築の際に、コンポーネントの作成と同時に求める。

2.2.3 アンビジテッドチルドレン $\mathcal{U}$

バケッティングを行うためには、あるコンポーネントを初めて visit したときに、その子供コンポーネントの $\min D$ がわかっている必要がある。この、まだ visit されていない (そして、visit された親を持つ) コンポーネントをアンビジテッドチルドレンと呼び、その $\min D$ を管理するためのデータ構造を $\mathcal{U}$ という。$\mathcal{U}$ の構築自体は $O(n)$ でできる。

$\mathcal{U}$ は、2 つの操作をサポートする。ひとつは、split 操作で、あるコンポーネントを visit するときのアンビジテッドチルドレンの更新に対応する。もうひとつは、change 操作で、ある点を visit して D が更新されたとき、その点を含むコンポーネントの $\min D$ の更新に対応する。split は最大 n 回発生し、トータル $O(n)$ で実現される。change は最大 m 回発生し、トータル $O(m)$ で実現される。

これら 2 つの操作を実行するために、Thorup はセグメント (コンポーネントの含む点を連続的区間で表したもの) とインターバル (2 分木) を用いている。詳細は省くが、上記の操作を線形時間で実行するために、インターバルは 2 段階構造を持っていて、1 段目では $\log n$ 個の頂点をひとまとめにして扱い、より細かく扱う必要があれば、2 段目 ($\log\log n$ 個ひとまとめ) に降りる。それ以上細かい処理は、$\log\log n$ 個の点をひとつの Q ヒープ ([FW94]) で扱うことで実現する。

2.3 アルゴリズム

1. MST を構築する。[FW94] の線形時間アルゴリズムを用いる。
   このアルゴリズムがアトミックヒープ、Q ヒープ、フィボナッチヒープを使用している。
2. MST を利用して $\mathcal{T}$ を構築する。
3. このとき同時に各コンポーネントのバケット幅を計算する。
4. $\mathcal{T}$ を利用して $\mathcal{U}$ を構築する。
5. **SSSP** をおこなう。

SSSP: SSSP 全体
Visit($\mathcal{T}$ のルートコンポーネント) して D を返す。

Expand($[v]_i$): バケットにコンポーネントを配置

1. $\mathcal{U}$ に、$[v]_i$を visit する事によって起こる split をおこなう。
2. 子供コンポーネントを、$\min D \downarrow i-1$ の値の位置のバケットに配置する。
3. バケットの先頭インデックス ix_0を $\min D([v]_i) \downarrow i-1$ にセット。

Visit($[v]_i$): コンポーネントを visit する

1. $i=0$ なら、**Visit**(v) をおこなって終了。
2. $\mathcal{T}$ 上の親のレベルを j とする。
3. 初めての visit なら **Expand** でバケット配置。
4. インデックス ix を ix_0にセット。
5. while コンポーネントが非空かつ $ix \downarrow j-1$ が増加しない間
   (a) このコンポーネントのバケット ix に子供があるなら、
   それらを順不同で **Visit** する。
   (b) ix をインクリメントする。
6. ix すなわち $\min D([v]_i^-) \downarrow j-1$ が増加したら、
   $[v]_i$の親のバケットでの位置が変わるので、正しく変更する。

Visit(v): 点を visit して D を更新する
Dijkstra のアルゴリズムと同様に、v に接続している点に関して D を小さくできるなら更新する。同時に、$\mathcal{U}$ の change をおこない、その点を含むコンポーネントのバケット再配置を行う。

なお、データ構造の構築終了後 (SSSP) は、**Visit**(v) がトータル $O(m)$、その他はトータルで $O(n)$ でできる。これは、コンポーネントの総数、バケットの総数がともに $O(n)$ であることによる。

3 実装に関する問題と変更

以上のアルゴリズムを現在の計算機上で実装することを考えると、MST および $\mathcal{T}, \mathcal{U}$ を構築する際に問題が生じる。ここでは、その問題とそれを解決するための変更点を提案する。

3.1 MST ([FW94]) の問題と変更

3.1.1 データ構造

MST の線形時間アルゴリズム ([FW94]) では、RAM を前提としたデータ構造、アトミックヒープと Q ヒープ (どちらも、サイズ制限はあるがヒープ操作をならし定数時間で実行できる。Q ヒープのほうがサイズ制限が厳しいが、ヒープ中の要素の順序も [AFK84] のプライオリティキュー同様に保持できる) が用いられているが、これらを実装する際に、以下のような問題が生じる。

1. アトミックヒープは、$n > 2^{12^{20}}$を前提としている。
2. Q ヒープは、非常に複雑なテーブルルックアップを用いている。

詳細は略すが、アトミックヒープは、フィボナッチヒープのように、ヒープのフォレストとして実現されている。そのフォレストのルートを管理する構造として、1個だけからなるQヒープを用いている。さらに、フォレストはその高さ(最大12)をインデックスとするバケツに配置されている。ひとつのバケットには、$(\log n)^{1/5}$のサイズ制限がある。したがって、フォレストの個数は最大で$12(\log n)^{1/5}$になりうる。しかし、1個のQヒープが扱える要素数は、最大$(\log n)^{1/4}$である。したがって、上記のような制限が生じる。

また、これも詳細は省くが、Qヒープはサイズ制限が$(\log n)^{1/4}$という非常に小さい値である上に、全ての操作を定数時間で行うために、非常に複雑で大きなテーブルルックアップを用いている。これは、メモリの面からもプログラムの面からも、非常に実装しにくい。

これらの問題を回避して、計算量を線形時間にしたまま実装するための提案を以下に示す。

1. アトミックヒープのバケット1個に1個のQヒープ(全部で12個)を割り当てる。
2. Qヒープはテーブルルックアップを使わず、単なる配列と比較操作で実現する。

まず、アトミックヒープのルートを、上記のように12個のQヒープで管理することによって、上記のサイズ制限を解除する。これによって、要素を挿入・削除したときの最小ノードの更新は、12個のQヒープの最小ノードのうち最小のものを選ぶことによって実現される。変更前は1個のQヒープだけを参照すれば良かったので、計算時間は12倍かかることになるが、定数倍なので、オーダーは変わらない。

Qヒープは、サイズ制限が$(\log n)^{1/4}$と非常に小さい。これは、32ないし64ビットの計算機では、高々2となる。したがって、これをサイズ2の配列と比較操作だけで実現しても、定数時間でできるということになる。

3.1.2 理論上の問題

このMSTの線形時間アルゴリズムは、[FT87]で述べられた、フィボナッチヒープを用いた多パスMSTアルゴリズムに基づいている。しかし、[FT87]の多パスMSTアルゴリズムには些細な理論上のミスがあり、論文通りの方法では、正確なMSTが構築できないことがわかった。ここでは、その部分を単純に引用して、訂正することにした。

[FT87] 引用部分 (p.612)

> *Connect to Starting Tree.* Delete an old T of minimum key from the heap. Set $key(T) = -\infty$.
> (中略)
> To finish the growth step, empty the heap and set $key(T) = \infty$ for every old tree T with finite key (these are the trees that have been inserted into the heap during the current growth step).

ここで、ヒープから取り出した最小の要素Tについてkeyの値を$-\infty$にセットしては、最後のkeyを∞に戻すことにはならない。したがって$key(T) = -\infty$のままで、このTにつながるような枝は、二度と選ばれず、正しいMSTにはならない。このような例は、簡単に作ることができる。したがって、「有限のkeyを持つTについて」ではなく、「いったんヒープに入ったT」を記憶しておき、そのkeyを全て∞に戻すことにすれば、正しいMSTを構築することができる。

3.2 $\mathcal{T}$の問題と変更

MSTの構築を除いた、$\mathcal{T}$を構築する際の問題とは、以下の通りである。

1. パックドマージングソート ($n < b$のとき) は実装が困難

2. 線形時間での find and union は実装が困難

このうち、バックドマージングソートは、$n > b$ と仮定すれば必要ない。また、線形時間での find and union ([GT85]) の代わりに、実装が簡単な $O(m \log^* n)$ の経路圧縮アルゴリズムを用いることにした。なお、ここでは、MST の枝だけを使うので、これは $O(n \log^* n)$ ということになる。これは、現在の計算機で使える n の範囲では、$\log^* n < 5$ となって線形時間と言っても良い。

3.3 $\mathcal{U}$ の問題と変更

$\mathcal{U}$ は、理論では 2 段階の木構造で実現し、2 段目の $\log\log n$ 個の頂点を 1 つの Q ヒープで扱っていた。しかし、32 ないし 64 ビットの計算機では、$\lfloor \log\log n \rfloor > \lfloor (\log n)^{1/4} \rfloor$ なので、これは不可能である。

そこで、3 段目の木構造を新たに追加し、$\log\log\log n$ 個の頂点を 1 つの Q ヒープで扱うことにした。32 ないし 64 ビットの計算機では、$\lfloor \log\log\log n \rfloor \leq 2$ となって、1 個の Q ヒープで扱うことができる。また、こうしても、全体を $O(m)$ で扱うことには変わりがない。しかし、実装は 2 段階のものより、複雑なものになる。

4 実装と実験

上記の変更を踏まえて、MST の線形時間アルゴリズムを含む、Thorup の線形時間 SSSP アルゴリズムを実装した。言語は、C++を用いた。今回作成したプログラムは、3000 行を超え、Dijkstra のアルゴリズムの実装と比べて非常に複雑なものになってしまったが、とりあえず正解を出すことができるようになった。しかし、すべてのバグを除去するのに時間がかかりすぎ、実験の規模が小さくなってしまった。さらに、参考として、MST のアルゴリズムとして、フィボナッチヒープを用いた Prim のアルゴリズムを使用した Thorup のアルゴリズムの実装も作成した。

以降に、この実装、Dijkstra のアルゴリズムの単純な実装、そしてフィボナッチヒープを用いた Dijkstra のアルゴリズムの実装のランダムグラフ上での実行時間を計測した実験の結果を述べる。なお、ランダムグラフは、枝 (i, j) (i, j は頂点) が存在する確率 p を与えて生成し、長さは乱数を用いた。なお、連結性のために、$(i, i+1)$ には必ず枝が存在することにした。実験には、Sun の Ultra30 を用いた。

表 1: 各アルゴリズムの実行時間と Thorup の各部の実行時間、$n = 10000$ (sec)

m	34892	59680	99771	134577	259071
Dijkstra(naive)	13.12	13.36	13.61	13.68	14.03
Dijkstra(F-heap)	0.40	0.47	0.59	0.66	0.91
Our Thorup(total)	2.11	3.17	5.57	8.08	17.95
MST	1.60	2.58	4.82	7.29	16.82
Data structures	0.18	0.20	0.25	0.22	0.26
Visiting	0.33	0.39	0.50	0.57	0.87
MST(Fheap-prim)	0.17	0.24	0.32	0.40	0.65
Thorup(Fheap-prim)	0.66	0.83	1.07	1.17	1.78

Data Structures は、MST を構築した後に、$\mathcal{T}, \mathcal{U}$ を構築・初期化するのに要した時間。
Visiting は、データ構造を完成させた後の、SSSP に要した時間。
MST(Fheap-prim) は、参考のフィボナッチヒープを用いた Prim アルゴリズムに要した時間。

Thorup(Fheap-prim) は、その MST アルゴリズムを代わりに用いた Thorup のアルゴリズム。
なお、グラフをファイルから読み込む時間は除いてある。

$n = 10000$ のときには、表 1 のように、Thorup のアルゴリズムは比較的グラフが疎のときには、Dijkstra よりは高速だが、全体にフィボナッチヒープを用いた Dijkstra(Dijkstra F-heap) よりかなり遅い。とくに、MST の線形時間アルゴリズムの部分で、時間がかかっている。実際、参考に作成したフィボナッチヒープを用いた Prim の MST アルゴリズムを用いたほうがここでは高速である。なお、Visiting のところだけを Dijkstra F-heap と比較すると、前者のほうがいくぶん高速である。

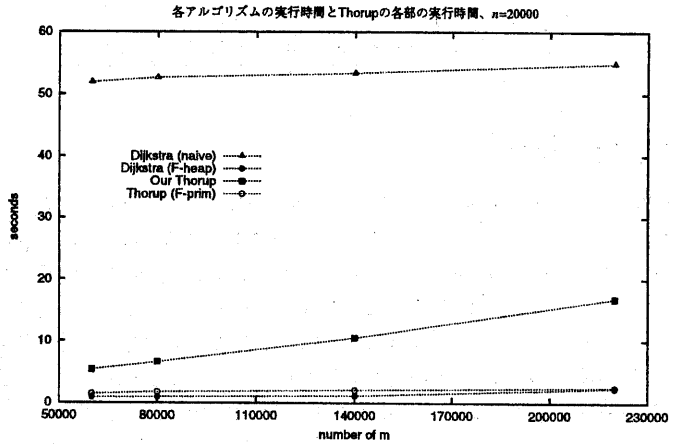

表 2: 各アルゴリズムの実行時間と Thorup の各部の実行時間、$n = 20000$ (sec)

m	59992	79950	140022	219910
Dijkstra(naive)	51.88	52.63	53.38	54.92
Dijkstra(F-heap)	0.84	0.93	1.12	2.34
Our Thorup(total)	5.37	6.59	10.49	16.86
MST	4.27	5.32	9.07	15.22
Data structures	0.41	0.52	0.44	0.48
Visiting	0.69	0.75	0.98	1.16
MST(Fheap-prim)	0.36	0.46	0.61	0.82
Thorup(Fheap-prim)	1.46	1.73	2.03	2.46

$n = 20000, 50000$ のときには、表 2,3(図は $n = 20000$ のみ) のように、やはり Thorup のアルゴリズムは比較的グラフが疎のときには、Dijkstra よりは高速だが、全体にフィボナッチヒープを用いた Dijkstra よりかなり遅い。全体の実行時間のうち、MST の時間が非常に大きな割合を占めているということもよりはっきりしてきている。なお、Visiting,Data structures は比較的高速で、MST の部分に Fheap-prim を用いると、全体がかなり高速になるということもわかる。

表 3: 各アルゴリズムの実行時間と Thorup の各部の実行時間、$n = 50000$ (sec)

m	175065	299914	424396
Dijkstra(naive)	327.53	326.63	326.97
Dijkstra(F-heap)	2.33	2.83	3.26
Our Thorup(total)	26.54	39.00	60.74
MST	23.33	35.24	56.60
Data structures	1.25	1.30	1.23
Visiting	1.96	2.46	2.91
MST(Fheap-prim)	1.13	1.61	2.03
Thorup(Fheap-prim)	4.34	5.37	6.17

5 まとめと今後の課題

結論として、上記のような変更を加えることによって、Thorup のアルゴリズム (MST の線形時間アルゴリズムを含む) を実装することが可能になったが、今回の実験に限れば、Thorup のアルゴリズムの実装はフィボナッチヒープを使った Dijkstra のアルゴリズムより遅く、実用的とは言えない。その主な原因は MST の線形時間アルゴリズムが実行時間の大部分を占めていることによるという結果も得られた。

さらに、Thorup のアルゴリズムのデータ構造は MST を含めて基本的に始点に依存しないデータ構造であるため、複数始点最短路問題では、始点が変わるたびにデータ構造を作り直す必要がないという事実も分かった。全始点最短路問題で各点ごとに繰り返される SSSP(Visiting) の部分は、今回の実験ではフィボナッチヒープを使った Dijkstra のアルゴリズムより少し高速であるから、より実装を工夫すれば、Thorup のアルゴリズムは実用的になる可能性があると考えられる。

また、今回の実験では、正確な実装の構築に時間がかかり、実験対象のグラフの規模が小さいものにとどまってしまった。今後は、より大規模なグラフについて同様の実験をおこない、今回の結果を補完したい。

参考文献

[AFK84] M. Ajtai, M. Fredman, and J. Komlos. Hash functions for priority queues. *Inf.Comput.*, 63: 217–225, 1984.

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J.ACM*, 34: 596–615, 1987.

[FW93] M. L. Fredman and D. E. Willard. Surpassing the imformation theoretic bound with fusion trees. *J.Comput.Syst.Sci.*, 47: 437–458, 1993.

[FW94] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J.Comput.Syst.Sci.*, 48: 533–551, 1994.

[GT85] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J.Comput.Syst.Sci.*, 30: 209–221, 1985.

[Tho97] M. Thorup. Undirected single source shortest paths in linear time. In *Proceedings of the 38th Symposium on Foundations of Computer Science(FOCS)*, pages 12–21, 1997.